BATCH ANOMALY DETECTION

August 2022

AUTHOR(S):
Eya ABID

IT-CD-CC

SUPERVISOR(S):
Martin ADAM
Jaroslava SCHOVANCOVÁ
Gavin McCANCE

ABSTRACT

The 300,000 CPU-core HTCondor Batch farm at CERN provides the computing power for the initial processing of data coming from the LHC experiments. Such a large-scale computing setup inevitably comes with an abundance of monitoring data; current monitoring methods cannot be configured in a reasonable amount of time to catch all the potential anomalies. Building on previous work done and ongoing in the IT department, this project will focus on the HTC batch system, using both the base monitoring metrics and the HTC job data to evaluate our options for better anomaly detection and handling.

TABLE OF CONTENTS

1 ACKNOWLEDGMENT

I would love to express my genuine gratitude towards my lead supervisor Martin ADAM for his continuous support, patience, and guidance throughout the whole internship period. I am also grateful for all the support and professionalism of the whole IT-CD-CC section during my stay.

2 INTRODUCTION

The CERN Batch Service is a fairly standard High Throughput Computing (HTC) Batch System with a fair-sharing mechanism. Its purpose is to allow users to queue up jobs in the system, and maximise the utilisation of the batch farm (currently around 100k cores) while respecting the agreed-upon fair-share policies set by CERN and experiment management [3].

With a resource pool as large and with workflows as diverse, the number of possible states the infrastructure can end up in is immense. There is an on-going effort to make the service administration automated.

The open question is, how to improve the sensors part of that effort. Are there workers in an erratic state for a significant amount of time without being fixed thus wasting resources? Can we flag a job as problematic in advance (i.e. before it gets to run)?

Questions like these might be solved by a machine learning approach, a manual approach is not feasible due to the breadth of the source domain.

This project will look into gathering already available data. This will mean using multiple input sources such as MONIT and some in-house data sources. Then this data will be compiled into a data-set usable as an input for a machine learning algorithm. After bench-marking several various approaches, we will recommend the best solution to augment the current batch automation project.

3 PROJECT OVERVIEW

3.1 CERN Monitoring Setup

The Data center Monitoring in the CERN IT is handled by the MONIT project [6].

For our purposes, we need to use the HW monitoring data collected by MONIT. We could either access the data directly on the MONITs HDFS, or use the new ADMON [1] project that promises to provide easier access to this data (besides many other useful features). The data are collected by the Collectd daemon, however MONIT has its own data format.

Data from jobs take the format of events whenever there is a change in the job status (Queuing, Launching, Terminating,...) an event is send via a Graphite based solution.

3.2 Analytical Platform(s)

There are multiple data processing platforms provided by other teams in the CERN IT.

The SWAN (Service for Web based ANalysis)[14] platform is a great tool to get access to our data and harness the power of a large Spark cluster from the comfort of our browser window.

There is also the Kubeflow based platform[4] dedicated for Machine Learning workflows at CERN. It is a better solution for even heavier or production-close workloads than SWAN, but it's also a bit more complicated.

The SWAN services were adapted for this project's workflow since it is in its initial stages.

4 PROJECT PROGRESS AND IMPLEMENTATION

4.1 Data Collection

Data Collecting and understanding is the building block to every data analysis pipeline. This is why we pledged a great percentage of the project timeline to fully comprehend the data we are dealing with.

4.1.1 ADMON For Data Fetching

Initially, we opted for the ADMON Python API to fetch and collect the monitoring data. ADMON is implemented to unify anomaly detection on data stored in the IT Monitoring Service at CERN. The way it works is you define the input data and the inference model and ADMON will scheduled get the data out from Kafka, preprocess it, send it to the model and send the results back to MONIT.

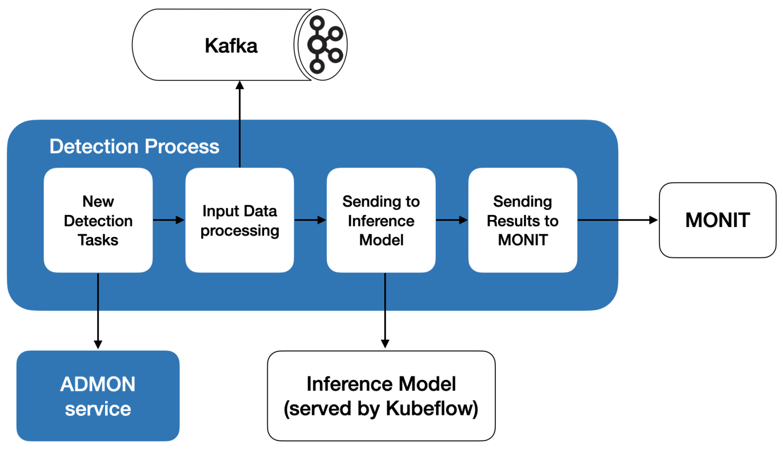
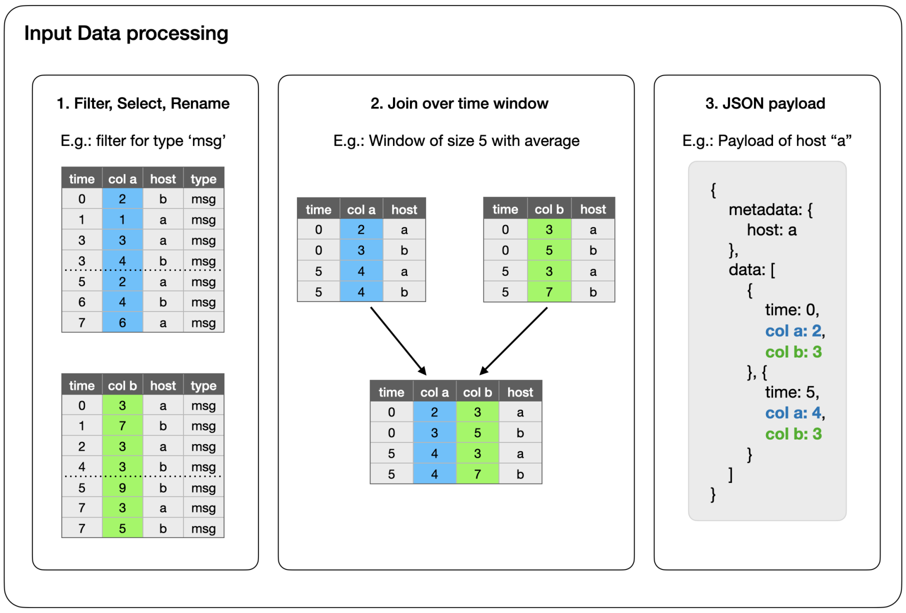


Figure 1: ADMON's Data Detection Pipeline

However, the outputted JSON file of the Python API was a bit restricting since it did not allow all the plugin metrics to be in one data-frame, which required multiple Join operation in order to merge each plugin JSON output into one table.


Input Data processing
1. Filter, Select, Rename
E.g.: filter for type ‘msg’
2. Join over time window
E.g.: Window of size 5 with average
3. JSON payload
E.g.: Payload of host “a”

time	col a	host	type
0	2	b	msg
1	1	a	msg
3	3	a	msg
3	4	b	msg
5	2	a	msg
6	4	b	msg
7	6	a	msg

time	col b	host	type
0	3	a	msg
1	7	b	msg
2	3	a	msg
4	3	b	msg
5	9	b	msg
7	3	a	msg
7	5	b	msg

time	col a	host
0	2	a
0	3	b
5	4	a
5	4	b

time	col b	host
0	3	a
0	5	b
5	3	a
5	7	b

time	col a	col b	host
0	2	3	a
0	3	5	b
5	4	3	a
5	4	7	b

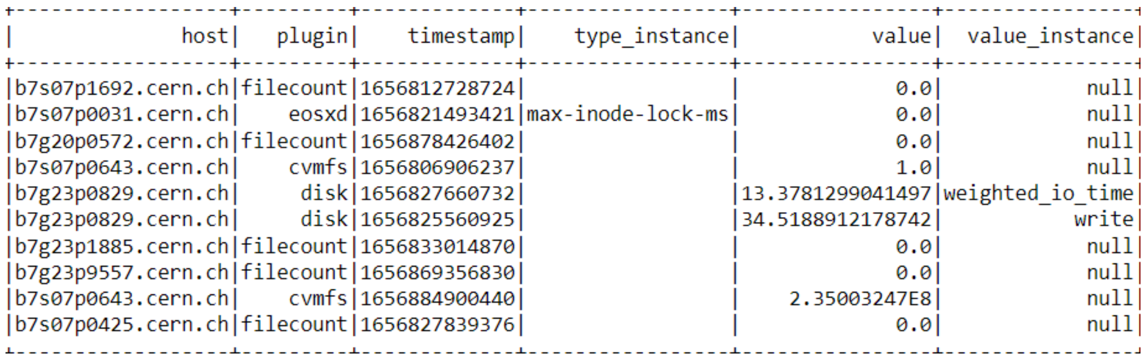
```
{
    metadata: {
        host: a
    },
    data: [
        {
            time: 0,
            col a: 2,
            col b: 3
        }, {
            time: 5,
            col a: 4,
            col b: 3
        }
    ]
}
```

Figure 2: ADMON’s Input Data Processing Pipeline [1]

As far as we know, the Join operation is quite consuming within the Spark environment, which required us to handle the monitoring raw data manually.

4.1.2 Handling Raw Data

The raw data was a challenge to deal with, and to make this data easier to manipulate within Spark, a choice of one host per query was adapted. After selecting the most relevant metrics, the outputted table is the following:



```
+------------------+---------+-------------+----------------+----------------+----------------+
|              host|   plugin|    timestamp|   type_instance|           value|  value_instance|
+------------------+---------+-------------+----------------+----------------+----------------+
|b7s07p1692.cern.ch|filecount|1656812728724|                |             0.0|            null|
|b7s07p0031.cern.ch|    eosxd|1656821493421|max-inode-lock-ms|             0.0|            null|
|b7g20p0572.cern.ch|filecount|1656878426402|                |             0.0|            null|
|b7s07p0643.cern.ch|    cvmfs|1656806906237|                |             1.0|            null|
|b7g23p0829.cern.ch|     disk|1656827660732|                |13.3781299041497|weighted_io_time|
|b7g23p0829.cern.ch|     disk|1656825560925|                |34.5188912178742|           write|
|b7g23p1885.cern.ch|filecount|1656833014870|                |             0.0|            null|
|b7g23p9557.cern.ch|filecount|1656869356830|                |             0.0|            null|
|b7s07p0643.cern.ch|    cvmfs|1656884900440|                |     2.35003247E8|            null|
|b7s07p0425.cern.ch|filecount|1656827839376|                |             0.0|            null|
+------------------+---------+-------------+----------------+----------------+----------------+
```

Figure 3: The Initial Fetched Data

The most important features are **value**, **type instance**, as well as the **plugin** column.

4.2 Data Manipulation

The goal now is to make this data in the proper schema to be fed to a machine learning model. This meant to ultimately split the values across the various instance types in order to create, for each plugin, the corresponding aggregated values, per host-group.

The focus was on the following instances:

- cpu-idle
- cpu-nice
- cpu-interrupt
- cpu-system
- cpu-steal
- cpu-user
- cpu-wait
- df-reserved
- df-free
- df-used
- load-long
- load-mid
- load-short
- proc-run
- proc-sleep
- proc-block
- proc-zombie
- swap-in
- swap-out
- swap-in
- net-in
- net-out

Regarding the aggregation type, whether summing or averaging the values, the **dstype** value is used to determine which operation to follow. Windowing the data via a 60-minute window was also adapted for further aggregation and to reduce the overall size of the data.

4.3 Data Mining

4.3.1 IsolationForest

The Isolation Forest Algorithm: [10] This algorithm returns the anomaly score of each sample.

The IsolationForest 'isolates' observations by randomly selecting a feature and then randomly selecting a split value between the maximum and minimum values of the selected feature.

Since recursive partitioning can be represented by a tree structure, the number of splittings required to isolate a sample is equivalent to the path length from the root node to the terminating node.

This path length, averaged over a forest of such random trees, is a measure of normality and our decision function.

Random partitioning produces noticeably shorter paths for anomalies. Hence, when a forest of random trees collectively produce shorter path lengths for particular samples, they are highly likely to be anomalies.

Implementation: After randomly splitting the data into test data and train data, scaling it, vectorizing the features, and normalizing them, we fitted the algorithm and outputted a total of 35 anomalies among 311 observations in the test set.

These anomalies are determined by peaks in the network plugin across a five-hour window. The other metrics did not show any peaks or abnormal behaviour.

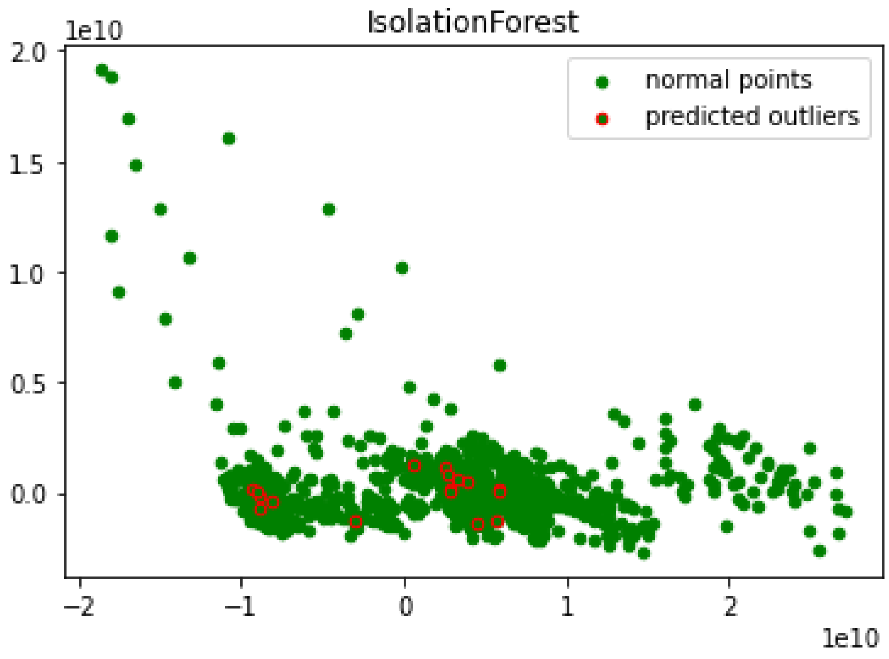


Figure 4: IsolationForest Detected Anomalies

4.3.2 LinearSVC

The LinearSVC Algorithm: In machine learning, support-vector machines (SVMs) are supervised learning models with associated learning algorithms that analyze data for classification and regression analysis.[5]

The LinearSVC algorithm performs classification using linear support vector machines. This binary classifier optimizes the Hinge Loss using the OWLQN optimizer and it only supports L2 regularization currently.[7]

Implementation: Since IsolationForest was an unsupervised machine learning algorithm, it needed no labels for the classification task. SVMs on the other hand, are supervised machine learning algorithms, which requires a pre-set label columns to guide the model through the learning process. The first step was to determine the class column, which means to generate our proper classification to anomalies. This method is prone to biased results, since our interpretation of anomalies is not accurate. We determined the anomalous host-groups via a threshold superior to the mean minus standard deviation if each plugin value.

The algorithm outputted a total of 30 anomalies for a single host-group on a single day, which is comparably less performing than the first unsupervised algorithm.

4.3.3 K-Means Algorithm

The K-Means Algorithm: [11] The K-Means algorithm clusters data by trying to separate samples in n groups of equal variance, minimizing a criterion known as the inertia or within-cluster sum-of-squares. This algorithm requires the number of clusters to be specified. It scales well to large numbers of samples and has been used across a large range of application areas in many different fields. K-means is often referred to as Lloyd's algorithm. In basic terms, the algorithm has three steps. The first step chooses the initial centroids, with the most basic method being to choose samples from the data-set . After initialization, K-means consists of looping between the two other steps. The first step assigns each sample to its nearest centroid. The second step creates new centroids by taking the mean value of all of the samples assigned to each previous centroid. The difference between the old and the new centroids are computed and the algorithm repeats these last two steps until this value is less than a threshold. In other words, it repeats until the centroids do not move significantly.

Implementation:

The first attempt: Without the Elbow Method: In order to give the algorithm the best possible results, an optimal number of clusters must be first given in the algorithm parameters.

Below is the result of k = 2:

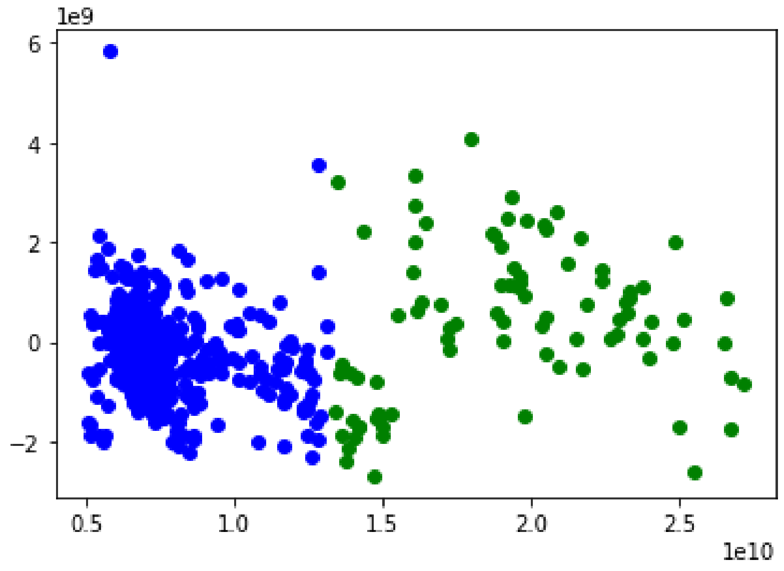


Figure 5: Clustering Result (k=2)

Below is the result of k = 4:

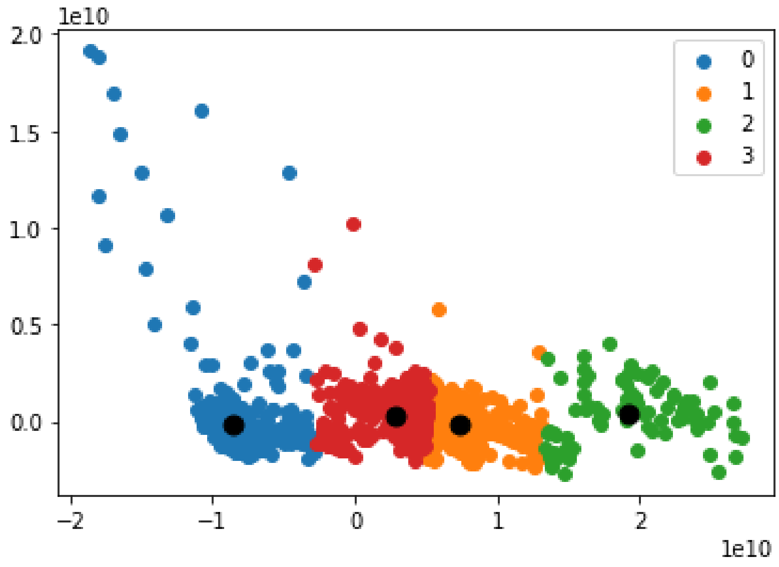


Figure 6: Clustering Result (k=4)

The first interpretation noticed is the ambiguity of the clustering criteria. The data points are not visibly split, which requires another method.

The second attempt: With the Elbow Method: The K-Elbow Visualizer implements the "elbow" method of selecting the optimal number of clusters for K-means clustering. K-means is a simple unsupervised machine learning algorithm that groups data into a specified number (k) of clusters. Because the user must specify in advance what k to choose, the algorithm is somewhat naive – it assigns all members to k clusters even if that is not the right k for the data-set.

The elbow method runs k-means clustering on the data-set for a range of values for k (say from 1-10) and then for each value of k computes an average score for all clusters. By default, the distortion score is computed, the sum of square distances from each point to its assigned center. Other metrics can also be used such as the silhouette score, the mean silhouette coefficient for all samples or the calinski-harabasz score, which computes the ratio of dispersion between and within clusters.

When these overall metrics for each model are plotted, it is possible to visually determine the best value for k. If the line chart looks like an arm, then the "elbow" (the point of inflection on the curve) is the best value of k. The "arm" can be either up or down, but if there is a strong inflection point, it is a good indication that the underlying model fits best at that point. [13]

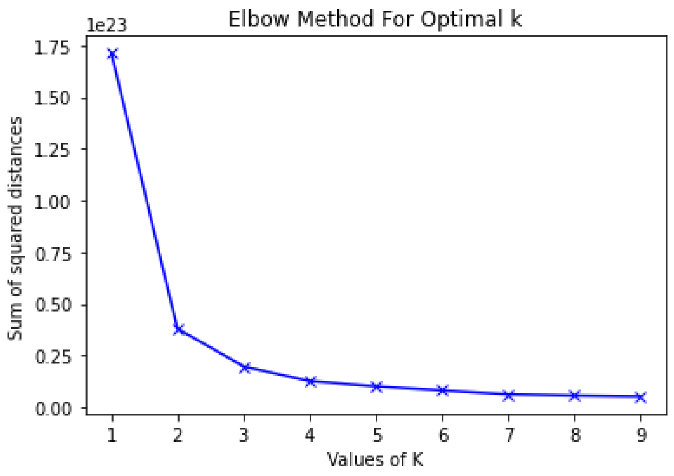


Figure 7: The Optimal Number of Clusters Using The Elbow Method

The figure above showcases that the optimal number of clusters is 3, which what we will adapt from now on using the k-means algorithm.

The third attempt: With PCA: Before applying the k-means algorithm again, we noticed that the data-set dimensions played a major role in making the algorithms slightly ambiguous, which led us to a method to reduce the dimensionality of the data: Principal component analysis (PCA).

Linear dimensionality reduction using Singular Value Decomposition of the data is used to project it to a lower dimensional space. The input data is centered but not scaled for each feature before applying the SVD.

It uses the LAPACK implementation of the full SVD or a randomized truncated SVD by the method of Halko , depending on the shape of the input data and the number of components to extract.[12]

After applying the PCA algorithm with a number of dimensions equal to 2, the projected data looked in the shape below:

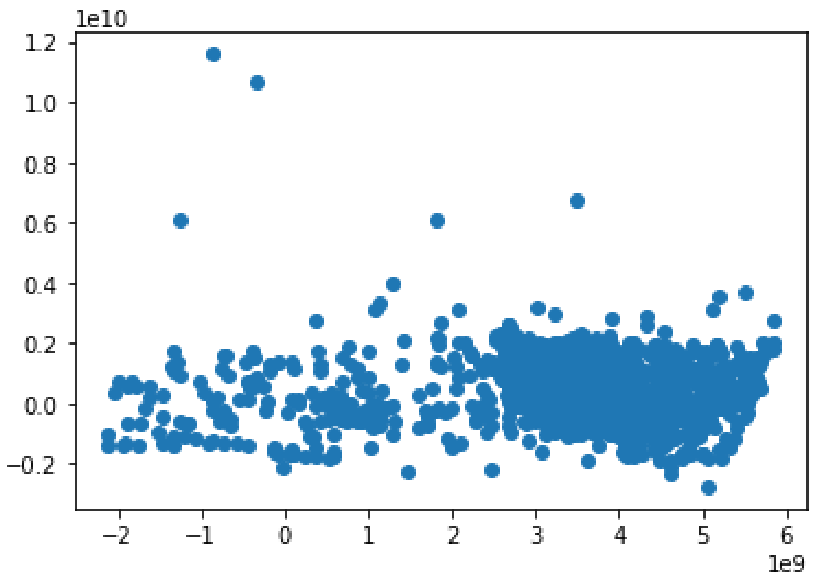


Figure 8: Data After PCA

As we can notice, the granularity has been reduced and the data points are visually more interpretable. If we split the data points into two clusters, we can easily spot that they are not equally distributed, and that a number of them is not following the pattern, which classifies them as outliers.

After applying the k-means algorithm on the reduced data, we see the following graph:

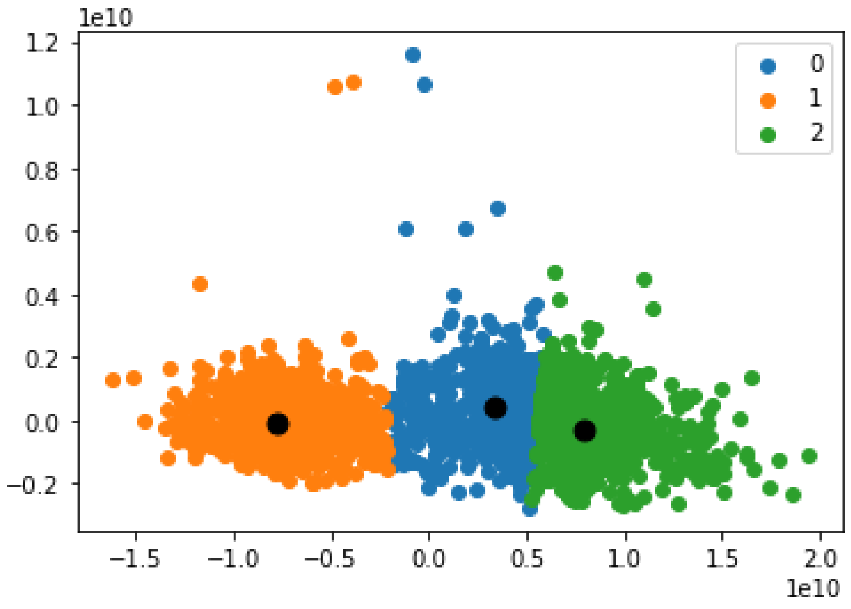


Figure 9: Clustering Result (k=3)

It is noticeable that two of the targeted clusters are meshed and rather should be merged into one cluster. We applied the algorithm again with k = 2, and labeled these clusters as outliers/inliers:

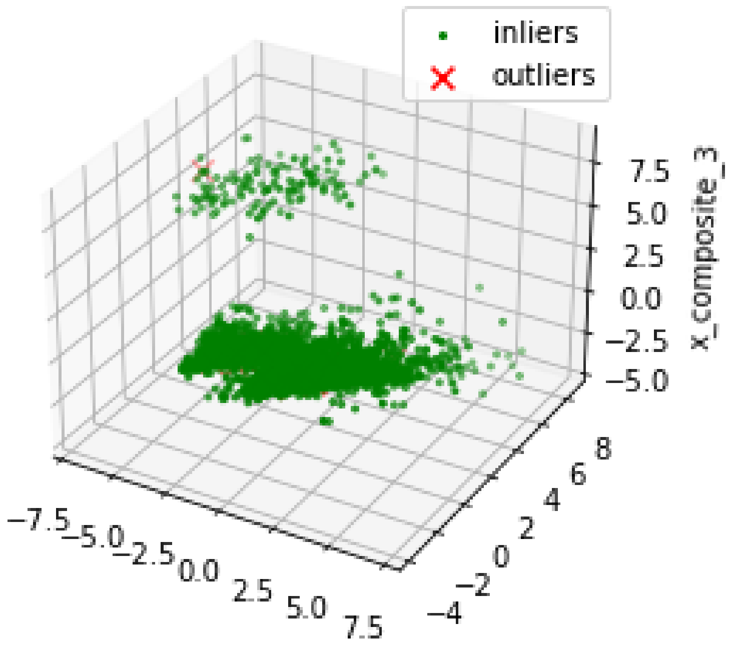


Figure 10: Clustering Result (k=2)

The algorithm detected one anomaly for a single host-group on a single day.

4.3.4 Auto-encoders - TensorFlow

Another way to reduce the dimensions of the data is using deep learning to regenerate these dimensions after reducing them. This is what we call Auto-encoders.

The Auto-encoders Algorithm: Auto-encoders are an unsupervised learning technique in which we leverage neural networks for the task of representation learning. Specifically, we'll design a neural network architecture such that we impose a bottleneck in the network which forces a compressed knowledge representation of the original input. If the input features were each independent of one another, this compression and subsequent reconstruction would be a very difficult task. However, if some sort of structure exists in the data (correlations between input features), this structure can be learned and consequently leveraged when forcing the input through the network's bottleneck.[2]

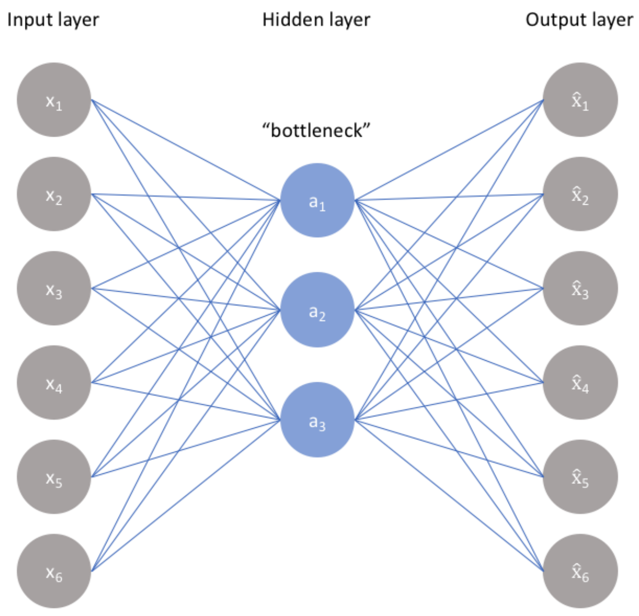


Figure 11: The Auto-encoder network structure

Implementation: We defined the model as the following:

```
class AnomalyDetector(Model):
    def __init__(self):

        super(AnomalyDetector, self).__init__()

        self.encoder = tf.keras.Sequential([
            layers.Dense(32, activation="relu"),
            layers.Dense(16, activation="relu"),
            layers.Dense(8, activation="relu")])

        self.decoder = tf.keras.Sequential([
            layers.Dense(8, activation="relu"),
            layers.Dense(16, activation="relu"),
            layers.Dense(32, activation="sigmoid")])

        def call(self, x):
            encoded = self.encoder(x)
            decoded = self.decoder(encoded)
            return decoded
```

Listing 1: The Auto-encoder code snippet used to define the corresponding model structure and layers.

We implemented both of the encoder and decoder within one class: the AnomalyDetector class. Each part has three dense layers, activated by the Relu Function:

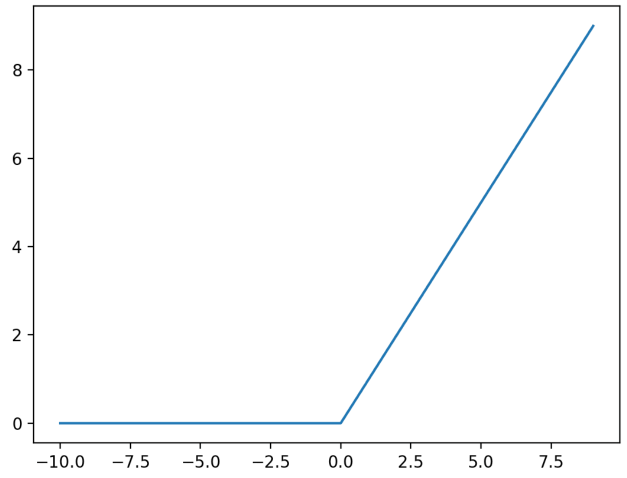


Figure 12: The ReLu Function [9]

After training the algorithm for 60 epochs, it was capable of regenerating the data as the following:

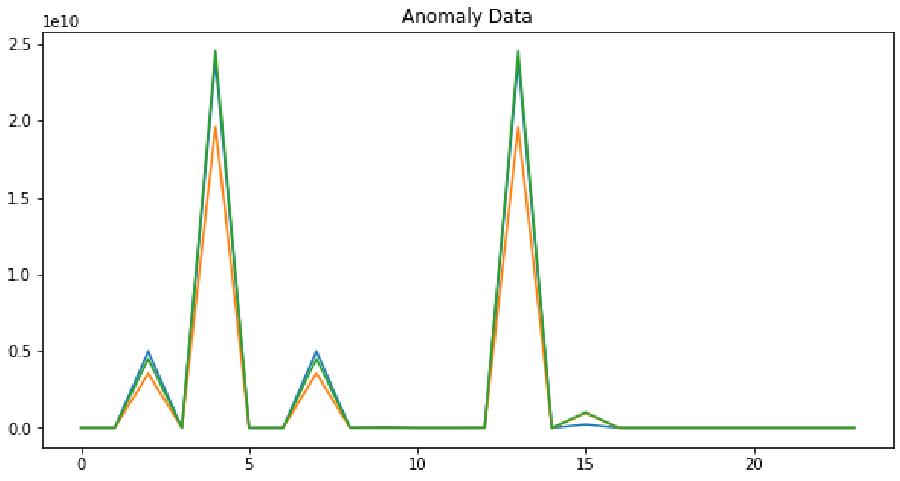


Figure 13: The Output of The Auto-encoder - Training Data Is In Green And Predicted Data Is In Orange

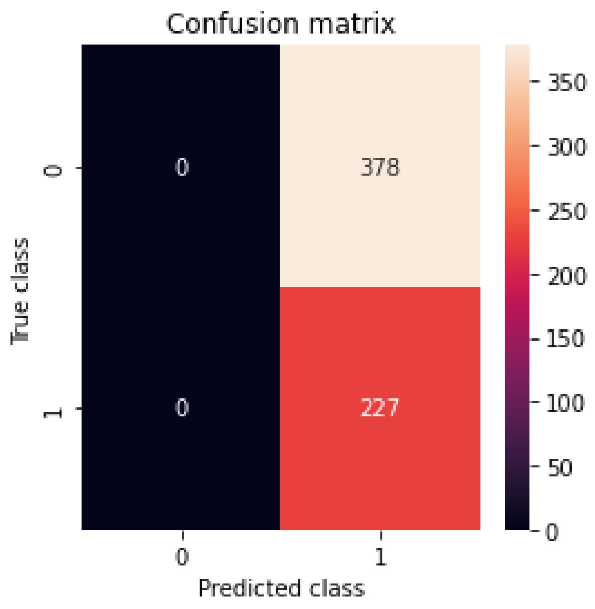


Figure 14: The Output of The Auto-encoder - The Confusion Matrix To Evaluate The Prediction Process

Even though the regeneration had an accuracy of 91%, the prediction process had only a 53% precision percentage, which is due to the non-diversity of the data. The data is not normally distributed, but it is rather linearly correlated which implies that we do not have a lot of data points, but rather a lot of copies of one data point.

4.3.5 Regressive Prediction

Seeing the linear correlation between the various plugin values, we decided to experiment with the Linear Regression algorithm, which is native in Spark.

Descriptive Analysis: Before implementing any regression algorithm, it is always a good habit to apply a series of analysis techniques to better understand the relationships between the data columns.

First, we plot the scatter matrix to spot which columns are mostly correlated to one another:

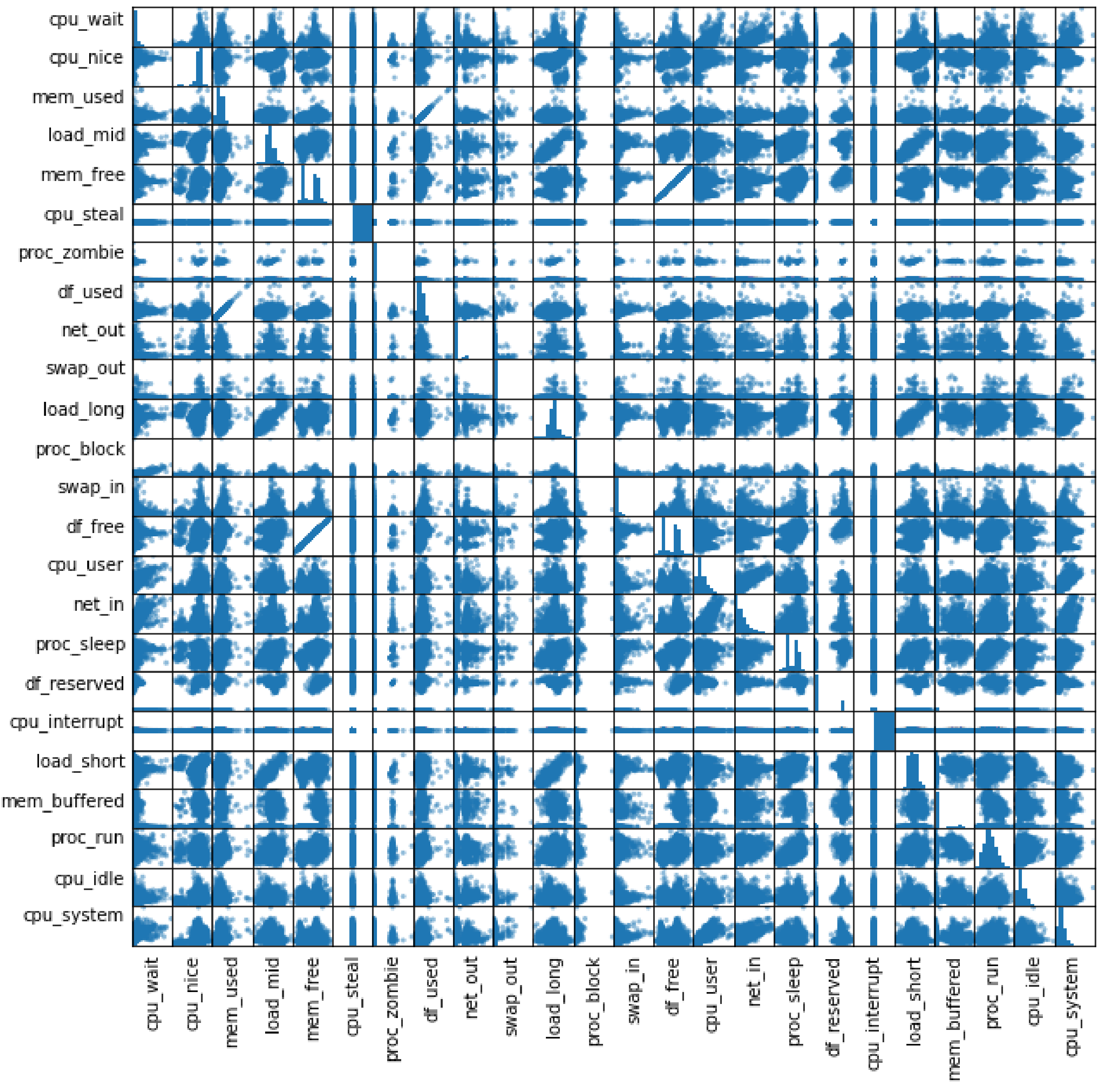


Figure 15: The Scatter Matrix Of The Plugin Data

It is easily noticeable that the same plugin metrics are highly correlated, whereas there is no inter-correlation between the plugin data.

Implementation: To better apply the Regression algorithm, we created two data-sets: df1 and df2. These data-sets are the historic one where we keep the past time windows, and the predictive one, or the one we will predict the plugin values through the regression algorithm.

We calculate RMSE which equals to 0.7980% on the training data. On test data, RMSE is equal to 0.784031%, which is slightly worse than the training data.

Root Mean Squared Error (RMSE) measures the differences between predicted values by the model and the actual values. However, RMSE alone is meaningless until we compare with the actual target variable value, such as mean, min and max. After such comparison, our RMSE looks pretty good.

Decision tree regression: To further improve the targeted RMSE, we implemented the Decision tree regression algorithm to obtain 0.403348% on the test data.

Gradient-boosted tree regression: This estimator builds an additive model in a forward stage-wise fashion; it allows for the optimization of arbitrary differentiable loss functions. In each stage a regression tree is fit on the negative gradient of the given loss function. [8]

This algorithm provided an RMSE of 0.369414% on the test data.

4.4 Bench-marking and Comparison

Due to the high correlation between the data columns, and to the linear aspect of the data distribution, supervised classification algorithms such the Auto-encoders and SVMs performed poorly with an average error rate of 45%.

On the other hand, unsupervised classification techniques such the Isolation-Forest, and the k-means (boosted with PCA) performed significantly better with an average error rate of 15%

The state-of-the-art models are the boosted linear models with an average error of 1%.

5 CONCLUSION

The batch anomaly detection pipeline based on raw monitoring data was implemented successfully using the Spark outline and SWAN infrastructure making the data model-feedable. A handful of anomaly detection techniques were implemented and tested on this data providing useful insights regarding the potential performance of the state-of-the-art algorithms making this pipeline scalable for more voluminous and complex data. Moreover, a data encoding-decoding technique was applied to further compress/decompress data in order to further reuse it without worrying about the hardware requirements nor the model limitations.

6 REFERENCES

[1] ADMON. URL: https://admon.docs.cern.ch.

[2] Auto-encoders. URL: https://www.jeremyjordan.me/autoencoders/.

[3] BATCH. URL: https://ccops.docs.cern.ch/batch/condor/index.html.

[4] KubeFlow. URL: https://ml.docs.cern.ch.

[5] LinearSVC. URL: https://en.wikipedia.org/wiki/Support-vector_machine#Linear_SVM.

[6] MONIT. URL: https://monit.web.cern.ch.

[7] LinearSVC in PySpark. URL: https://spark.apache.org/docs/latest/api/python/reference/api/pyspark.ml.classification.LinearSVC.html.

[8] Gradient-boosted Tree Regression. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.GradientBoostingRegressor.html.

[9] ReLu. URL: https://machinelearningmastery.com/rectified-linear-activation-function-for-deep-learning-neural-networks.

[10] IsolationForest in Sklearn. URL: https://scikit-learn.org/stable/modules/generated/sklearn.ensemble.IsolationForest.html.

[11] K-Means in Sklearn. URL: https://scikit-learn.org/stable/modules/clustering.html#k-means.

[12] PCA in Sklearn. URL: https://scikit-learn.org/stable/modules/generated/sklearn.decomposition.PCA.html.

[13] The Elbow Method in Sklearn. URL: https://www.scikit-yb.org/en/latest/api/cluster/elbow.html.

[14] SWAN. URL: https://swan.web.cern.ch/swan/.